Quincy: Fair Scheduling for Distributed Computing Clusters

Diksha Verma, Anjli Tyagi, Deepak Sharma
Department of Information Technology, Dronacharya college of Engineering

***Abstract*- This paper addresses the problem of scheduling concurrent jobs on clusters where application data is stored on the computing nodes. This setting, in which scheduling computations close to their data is crucial for performance, is increasingly common and arises in systems such as MapReduce, Hadoop, and Dryad as well as many grid-computing environments. We argue that data intensive computation benefits from a fine-grain resource sharing model that differs from the coarser semi-static resource allocations implemented by most existing cluster computing architectures. The problem of scheduling with locality and fairness constraints has not previously been extensively studied under this model of resourcesharing.**
We introduce a powerful and flexible new framework for scheduling concurrent distributed jobs with fine-grain resource sharing. The scheduling problem is mapped to a graph datastructure, where edge weights and capacities encode the competing demands of data locality, fairness, and starvation-freedom, and a standard solver computes the optimal online schedule according to a global cost model. We evaluate our implementation of this framework, which we call Quincy, on a cluster of a few hundred computers using a varied workload of data- and CPU-intensive jobs. We evaluate Quincy against an existing queue-based algorithm and implement several policies for each scheduler, with and without fairness constraints.
Quincy gets better fairness when fairness is requested, while substantially improving data locality. The volume of data transferred across the cluster is reduced by up to a factor of 3.9 in our experiments, leading to a throughput increase of up to 40%.
Categories and Subject Descriptors
D.4.1 [Operating Systems]: Process Management—Scheduling

***Index Terms*-Table 1: Job running time. The table shows the same data as Figure 1**
but here presented as the percentage of jobs under a particular running
time in minutes.
Table 1: Job running time. The table shows the same data as Figure 1
but here presented as the percentage of jobs under a particular running
time in minutes.

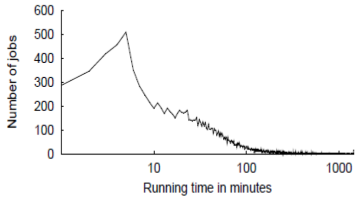
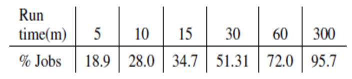


Figure 1: Distribution of job running times from a production cluster used inside Microsoft's search division. The horizontal axis shows the running time in minutes on a log scale, and the vertical axis shows the number of jobs with the corresponding running time.



Run time(m)	5	10	15	30	60	300
% Jobs	18.9	28.0	34.7	51.31	72.0	95.7

I. INTRODUCTION

Data-intensive cluster computing is increasingly important for a large number of applications including webscale data mining, machine learning, and network traffic analysis. There has been renewed interest in the subject since the publication of the MapReduce paper describing a large-scale computing platform used at Google . One might imagine data-intensive clusters to be used mainly for long running jobs processing hundreds of terabytes of data, but in practice they are frequently used for short jobs as well. For example, the average completion time of a MapReduce job at Google was 395 seconds
during September 2007 [8]. Figure 1 and Table 1 show data taken from a production cluster used by Microsoft's search division logging the duration of every job submitted over a period of 39 days from September 2008 to November 2008. While there are large jobs that take more than a day to complete, more than 50% of the jobs take less than 30 minutes. Our users generally strongly desire some notion of fair sharing of the cluster 1 resources. The most common request is that one user's
large job should not monopolize the whole cluster, delaying the completion of everyone else's (small) jobs. Of course, it is also important that ensuring low

latency for short jobs does not come at the expense of the overall throughput of the system. Section 5.2 sets out the exact definition of unfairness that we use to evaluate our system. Informally, however, our goal is that a job which takes t seconds to complete given exclusive access to a cluster should require no more than Jt seconds of execution
time when the cluster is shared among J concurrent jobs. Following Amdahl's law, most jobs cannot continuously make use of the resources of the entire cluster, so we can hope that many jobs will complete in fewer than Jt seconds. Many of the problems associated with data-intensive computing have been studied for years in the grid and parallel database communities. However, a distinguishing feature of the data-intensive clusters we are interested in is that the computers in the cluster have large disks directly
attached to them, allowing application data to be stored on the same computers on which it will be processed. Maintaining high bandwidth between arbitrary pairs of computers becomes increasingly expensive as the size of a cluster grows, particularly since hierarchical networks are the norm for current distributed computing clusters . If computations are not placed close to their input data, the network can therefore become a bottleneck. Additionally, reducing network traffic simplifies capacity planning. If a job is always allocated a certain fraction of the cluster's computing resources then ideally its running time should remain approximately the same. When concurrent jobs have high cross-cluster network
traffic, however, they compete for bandwidth, and modeling this dynamic network congestion greatly complicates performance prediction. For these reasons, optimizing the placement of computation to minimize network traffic is a primary goal of a data-intensive computing platform. The challenge for a scheduling algorithm in our setting is that the requirements of fairness and locality often conflict.
Intuitively this is because a strategy that achieves optimal data locality will typically delay a job until its
ideal resources are available, while fairness benefits from allocating the best available resources to a job as soon as possible after they are requested, even if they are not the resources closest to the computation's data. This paper describes our experience comparing a set of schedulers and scheduling policies that we operate on clusters containing hundreds of computers. The clusters are shared between tens of users and execute multiple jobs concurrently. Our clusters run the Dryad distributed execution engine, which has a similar lowlevel computational model to those of MapReduce, Hadoop and Condor . We believe that the Dryad computational model is well adapted to a fine-grain resource sharing strategy where every computer in the cluster is in general multiplexed between all of the running jobs. This is in contrast to traditional grid and highperformance computing models in which sets of cluster computers are typically assigned for exclusive use by a particular job, and these assignments change only rarely.
We examine this argument in detail in Section 2.1. The main contribution of this paper is a new, graphbased framework for cluster scheduling under a fine grain cluster resource-sharing model with locality constraints. We discuss related work in Section 7, however the problem of scheduling under this resource-sharing model has not been extensively studied before, particularly when fairness is also a goal. We show for the first time a mapping between the fair-scheduling problem for cluster computing and the classical problem of min-cost flow in a directed graph. As we demonstrate in Section 6, the global solutions found by the min-cost flow algorithm substantially outperform greedy scheduling approaches.
The graph-based formulation is also attractive from a software-engineering standpoint, since scheduling policies and tuning parameters are specified simply by adjusting weights and capacities on a graph datastructure, analogous to a declarative specification. In contrast, most previous queue-based approaches encode policies using heuristics programmed as imperative subroutines and this
can lead to increased code complexity as the heuristics become more sophisticated.
The structure of this paper is as follows. Section 2 sets out the details of our computing clusters and the
type of jobs that they are designed to execute. Section 3 describes several traditional queue-based scheduling algorithms that we use as a baseline for our experiments. Section 4 introduces the mapping between scheduling and min-cost flow, and explains the design of the Quincy scheduler in detail. In Sections 5 and 6 we outline our experimental design and evaluate the performance of the system on a medium-sized cluster made up of several hundred computers. After reviewing related work in Section 7, we conclude with a discussion of lessons learned, limitations of the approach, and opportunities for futur work.

II. THE PROBLEM SETTING

This section describes our computing environment, the type of jobs we execute, the definition of fairness we adopt, and the broad class of scheduling algorithms we consider. Not all distributed jobs or scheduling policies fit into our framework, but we postpone a discussion of how our system might be generalized until Section 8. We assume a homogeneous computing cluster under a single administrative domain, with many users competing for the cluster resources. There is a hierarchical network in which each rack contains a local switch and the rack switches are interconnected via a single core switch, so communication between computers in the same rack is “cheaper” than communication between racks. Our methods extend trivially to networks with no hierarchy (where the cost of communication is approximately equal between any pair of computers) and those with a deeper hierarchy.

2.1 Computational model

Our clusters are used to run jobs built on the Dryad distributed execution platform [17, 31]. Each job is managed by a “root task” which is a process, running on one of the cluster computers, that contains a state machine managing the workflow of that job. The computation for the job is executed by “worker tasks” which are individual processes that may run on any computer. A worker may be executed multiple times, for example to recreate data lost as a result of a failed computer, and will always generate the same result. This computational model is very similar to that adopted by MapReduce , Hadoop and Condor and the ideas in this paper should be readily applicable to those systems. A job’s workflow is represented by a directed acyclic graph of workers where edges represent dependencies, and the root process monitors which tasks have completed and which are ready for execution. While running, tasks are independent of each other so killing one task will not impact another. This independence between tasks is in contrast to multi-processor approaches such as coscheduling and programming models like MPI in which tasks execute concurrently and communicate during their execution. In fact, the data-intensive computational model leads to a somewhat different approach to resource sharing compared to traditional high performance computing clusters. When a cluster is used to execute MPI jobs, it makes sense to devote a specific set of computers to a particular job, and change this allocation only infrequently while the job is running, for example when a computer fails or a higher priority job enters the system. This is because MPI jobs are made up of sets of stateful processes communicating across the network, and killing or moving a single process typically requires the restart of all of the other processes in the set. In addition, high-performance computing clusters traditionally do not have a large quantity of direct-attached storage so while it may be advantageous to ensure that the processes in a job are scheduled close to each other in network topology, the exact set of computers they run on does not have a major effect on performance. It is therefore not usually worth moving a job’s processes once they have been allocated a fixed set of resources.

In contrast, workloads for systems such as Dryad and MapReduce tend to be dominated by jobs that process very large datasets stored on the cluster computers themselves. It generally makes sense to stripe each large dataset across all the computers in the cluster, both to prevent hot spots when multiple jobs are concurrently reading the same data and to give maximum throughput when a single job is executing. In addition, as noted above, tasks run independently so they can start and complete in a staggered fashion as resources become available, and killing a task only forfeits the work that task has done without affecting other running processes. Since tasks are staggered rather than executing in lockstep, most jobs fluctuate between periods in which they have many more ready tasks than there are available cluster resources, and periods in which a few straggler tasks are completing. The number of computers it is worth assigning to a job is therefore continuously changing. Consequently, a natural way to share a cluster between multiple Dryad jobs is to multiplex access to all the cluster computers across all running jobs, so a computer is in general assigned to a different job once its current task completes, rather than giving any job a long-term private allocation. This is the primary reason that new approaches such as the scheduling framework described in this paper are needed, and standard techniques such as gang scheduling are not applicable to our setting. In this paper we consider two broad techniques for giving a job its fair share of the cluster’s resources: running the job’s worker tasks in sub-optimal locations instead of waiting for the ideal computer to be available, which reduces latency but increases network traffic; and killing running tasks of one job to free resources for another job’s tasks, which can improve the latter job’s latency without sacrificing its data locality but wastes the work of the preempted task. Since both of these strategies can harm overall throughput, there is potentially a penalty to be paid for improving fairness. As we show in Section 6, scheduling algorithms that optimize a global cost function do a better job of managing this tradeoff than those which rely on greedy heuristics.

2.2 Cluster architecture

Figure 2 shows an outline of our cluster architecture.There is a single centralized scheduling service running in the cluster maintaining a batch queue of jobs. At any given time there may be several concurrent jobs sharing the resources of the cluster and others queued waiting for admission. When a job is started the scheduler allocates a computer for its root task. If that computer fails the job will be re-executed from the start. Each running job’s root task submits its list of ready workers, and their input

data summaries as defined below in Section 2.3, to the scheduler. The scheduler then matches tasks to computers and instructs the appropriate root task to set them running. When a worker completes, its root task is informed and this may trigger a new set of ready tasks to be sent to the scheduler. The root task also monitors the execution time of worker tasks, and may for example submit a duplicate of a task that is taking longer than expected to complete . When a worker task fails because of unreliable cluster resources, its root task is responsible for back-tracking through the dependency graph and resubmitting
tasks as necessary to regenerate any intermediate data that has been lost. The scheduler may decide
to kill a worker task before it completes in order to allow other jobs to have access to its resources or to move the worker to a more suitable location. In this case the scheduler will automatically instruct the worker's root so the task can be restarted on a different computer at a later time.
The scheduler is not given any estimates of how long workers will take, what resources they will consume, or given insight into the upcoming workers that will be submitted by jobs once running tasks complete. Section 8 discusses ways in which our system could be enhanced if such information were available.

2.3 Data locality

A worker task may read data from storage attached to computers in the cluster. These data are either inputs to the job stored as (possibly replicated) partitioned files in a distributed file system such as GFS or intermediate output files generated by upstream worker tasks. A worker is not submitted to the scheduler until all of its input files have been written to the cluster, at which time their sizes and locations are known. Consequently the scheduler can be made aware of detailed information about the data transfer costs that would result from executing a worker on any given computer. This information
is summarized as follows. When the nth worker in job j, denoted wjn is ready, its
root task rj computes, for each computer m, the amount of data that
would have to read across the network if it were executed on m. The root rj then constructs
two lists for task wjn : one list of preferred computers and one of preferred racks. Any computer that stores more than a fraction of wjn total input data is added to the first list, and any rack whose computers in sum store more than a fraction of wjn's total input data is added to the second. In practice we find that a value of 0:1 is effective for both _c and _r. These settings ensure that each list has at most ten entries and avoids marking every computer in the cluster as "preferred" when a task reads a large number of approximately equal-sized inputs, for example when a dataset is being re-partitioned. Some schedulers, including our flow-based algorithms, can make use of fine-grain information about the number
of bytes that would be transferred across the network as a result of scheduling wjn
on computer m. The precise information we send to the Quincy scheduler is described
in the Appendix.

2.4 Fairness in a shared cluster

As stated in the introduction, our goal for fairness is that a job which runs for t seconds given exclusive access to a cluster should take no more than Jt seconds when there are J jobs concurrently executing on that cluster. In practice we only aim for this guarantee when there is not heavy contention for the cluster, and we implement admission control to ensure that at most K jobs are executing at any time. When the limit of K jobs is reached subsequent jobs are queued and started, in order of submission time, as running jobs complete. During periods of contention there may therefore be a long delay between a job's submission and its completion, even if its execution time is short. Setting K too large may make it
hard to achieve fairness while preserving locality, since it increases the likelihood that several jobs will be competing for access to data stored on the same computer. However with too small a value of K some cluster computers may be left idle if the jobs do not submit enough tasks. Section 8 briefly discusses possible mechanisms for setting K adaptively. A job is instantaneously allocated a fraction _ of the
cluster by allowing it to run tasks on _M of the cluster's M computers. The allocated fraction _ can change over the running time of the job as other jobs enter and leave the system, as can the particular computers that the job'stasks are scheduled on. Of course this is a simplified model of sharing. We do not currently attempt to give jobs a fair share of network resources, or account for the resources consumed when a job's task reads remote data from a computer that is assigned to another job. Also, we currently restrict each computer to run only one task at a time. This is a conservative policy that is partly determined
by the fact that jobs do not supply any predictions about the resource consumption of their workers. Section 8 discusses ways of relaxing some of these restrictions. Our implementation aims to give each running job an equal share of the cluster. However, the methods we describe can straightforwardly be extended to devote any desired fraction of the cluster to a given job, and a system could implement priorities and user quotas by choosing these fractions appropriately. In the case that controlling

end-to-end job latency is important even when the cluster is suffering contention, one could implement a more sophisticated queueing policy, for example to let jobs that are predicted to complete quickly "jump the queue" and begin execution ahead of jobs that are expected to be long-running.

III. QUEUE-BASED SCHEDULING

This section describes a number of variants of a queuebased scheduling approach that we will use as a baseline for comparisons with our new flow-based framework. It seems natural to use a set of queues as the fundamental datastructure for a locality-based scheduler. The Hadoop scheduler adopts this model [1] and, while there are no published details of the MapReduce scheduler [7], it appears that queue-based approaches are the standard design for the public state of the art. (As far as we know, the Condor project [29] does not include any schedulers that make use of fine-grain locality information.) We include fairly sophisticated variants of the queuebased approach that, we believe, extend previously published
work. This exhaustive treatment is an attempt to provide a fair comparison between queue-based scheduling and flow-based scheduling as organizing frameworks, rather than concentrating on a particular implementation. We are familiar with queue-based schedulers since they have been used extensively by Dryad in the past, and this section surveys our good-faith attempt to implement effective
fair scheduling in this model.

3.1 Outline of a queue-based architecture

The schedulers in this section maintain a number of queues as illustrated in Figure 3: one for each computer m in the cluster (denoted Cm), one for each rack l (denoted Rl), and one cluster-wide queue denoted X. When a worker task is submitted to the scheduler it is added to the tail of multiple queues: Cm for each computer on its preferred computer list, Rl for each rack l on its preferred rack list, and X. When a task is matched to a computer using one of the algorithms below it is removed from all the queues it had been placed in. This family of algorithms ignores exact data transfer sizes and treats all
preferred computers as equal. When a new job is started its root task is assigned a computer at random from among the computers that are not currently executing root tasks, and any worke task currently running on that computer is killed and reentered into the scheduler queues as though it had just
been submitted. K must be small enough that there are at least K + 1 working computers in the cluster in order that at least one computer is available to execute worker tasks.

3.2 Baseline algorithm without fairness

The simplest algorithm we consider schedules worker tasks independent of which job they come from, so there is no attempt to achieve fairness. Whenever a computer becomes free, the first ready task on Cm, if any, is dispatched to m. If Cm does not have any ready tasks, then the first ready task on Rl is dispatched to m, where l is m's rack. If neither Cm nor Rl contains a ready task then the first ready task, if any, on X is dispatched to m. We refer to this algorithm as "G" for "greedy." This algorithm aims to keep all computers busy at all times as long as there are worker tasks available to run. It will always try to match tasks to their preferred computers or racks where possible, in order of task submission. This can have two adverse effects on latency: if a job submits a large number of tasks on every computer's queue 5 then other jobs will not execute any workers until the first job's tasks have been run; and in a loaded cluster a task that has no preferred computers or racks may wait for a long time before being executed anywhere since there will always be at least one preferred task ready for every computer or rack.

3.3 Simple greedy fairness

A simple concept of fairness can be used to reduce the chance that a job that starts up and submits a large number of workers to all the scheduler's queues will starve subsequent jobs from receiving resources on which to run their tasks. We implement the same fairness scheme as the Hadoop Fair Scheduler [1]. In our implementation it is achieved by "blocking" a job, which is defined as follows.
When a job is blocked its waiting tasks will not be matched to any computers, thus allowing unblocked jobs to take precedence when starting new tasks. The matching procedure in Section 3.2 above pulls the first "ready" task from a queue. In the simple greedy algorithm every task in any queue is always considered to be ready. In order to implement simple fairness we instead define a task to be ready only if its job is not blocked. For now we do not consider killing running tasks, so any workers that have already been started when a job becomes blocked continue to run to completion. Now, whenever a computer becomes free and we want to match a new task to it, we must first determine which jobs are blocked. At this time each job is given an instantaneous allocation corresponding to the number of computers it is currently allowed to use.

3.4 Fairness with preemption

The problem of a job "hogging" the cluster with alarge number of long-running tasks can be addressed by

a more proactive approach to fairness: when a job j is running more than Aj workers, the scheduler will kill its over-quota tasks, starting with the most-recently scheduled task first to try to minimize wasted work. We refer to this algorithm as "GFP." As long as a job's allocation never drops to zero, it will continue to make progress even with preemption enabled, since its longest-running task will never be killed
and will therefore eventually complete. To achieve this starvation-freedom guarantee the cluster must contain at least 2K working computers if there are K concurrent jobs admitted: one for each job's root task and one to allocate to a worker from each job.

3.5 Sticky slots

One drawback of simple fair scheduling is that it is damaging to locality. Consider the steady state in which each job is occupying exactly its allocated quota of computers. Whenever a task from job j completes on computer m, j becomes unblocked but all of the other jobs in the system remain blocked. Consequently m will be assigned to one of j's tasks again: this is referred to as the "sticky slot" problem by Zaharia et al. [32] because m sticks to j indefinitely whether or not j has any waiting tasks that have good data locality when run on m. A straightforward solution is to add some hysteresis, and we implement a variant of the approach proposed by Zaharia et al.

IV. FLOW-BASED SCHEDULING

As queue-based scheduling approaches are extended to encompass fairness and preemption, the questions of which of a job's tasks should be set running inside it quota Aj , or which should be killed to make way for another job, become increasingly important if we wish to achieve good locality as well as fairness. In the previous section we adopted heuristics to solve these problems,
based around a greedy, imperative approach that considered a subset of the scheduler's queues at a time, as each new task arrived or left the system. In this section we introduce a new framework for concurrent job scheduling. The primary datastructure used by this approach is a graph that encodes both the structure of the cluster's network and the set of waiting tasks along with their locality metadata. By assigning appropriate weights and capacities to the edges in this graph, we arrive at a declarative description of our scheduling policy. We can then use a standard solver to convert this declarative policy to an instantaneous set of scheduling assignments that satisfy a global criterion, considering
all jobs and tasks at once. The primary intuition that allows a graph-based declarative description of our problem is that there is a quantifiable cost to every scheduling decision. There is a data transfer cost incurred by running a task on a particular computer; and there is also a cost in wasted time to killing a task that has already started to execute. If we can at least approximately express these costs in the same
units (for example if we can make a statement such as "copying 1GB of data across a rack's local switch costs the same as killing a vertex that has been executing for 10 seconds") then we can seek an algorithm to try to minimize the total cost of our scheduling assignments. Having chosen a graph-based approach to the scheduling problem, there remains the question of exactly what graph to construct and what solver to use. In this paper we describe a particular form of graph that is amenable
to exact matching using a min-cost flow solver. We refer to this combination of graph construction and matching algorithm as the Quincy scheduler, and show in Section 6 that it outperforms our queue-based scheduler for every policy we consider.

4.1 Min-cost flow

We choose to represent the instantaneous scheduling problem using a standard flow network [11]. A flow network is a directed graph each of whose edges e is annotated with a non-negative integer capacity ye and a cost pe, and each of whose nodes v is annotated with an integer "supply" _v where

4.2 Encoding scheduling as a flow network

The scheduling problem described in Section 2 can be encoded as a flow network as shown in Figure 4 and explained in detail in the Appendix. This graph corresponds to an instantaneous snapshot of the system, encoding the set of all worker tasks that are ready to run and their preferred locations, as well as the running locations and current wait times and execution times of all currently-executing workers and root tasks. One benefit, and potential pitfall, of using a flow network formulation is that it is easy to invent complicated weights and graph structures to attempt to optimize for particular behaviors. For elegance and clarity we have attempted to construct the simplest possible graph, with the fewest constants to
set, that allows us to adjust the tradeoff between latency and throughput. The only hard design constraint we have adopted is starvation-freedom: we can prove as sketched in the Appendix that under weak assumptions every job will eventually make progress, even though at any moment some of its tasks may be preempted to make way for other workers. The overall structure of the graph can be interpreted
as follows. A flow of one unit along an edge in the graph can be thought of as a "token" that corresponds
to the scheduling assignment of one task. Each submitted worker or root task on the left hand side

receives one unit of flow as its supply. The sink node S on the right hand side is the only place to “drain off” the flow that enters the graph through the submitted tasks, so for a feasible flow each task must find a path for its flow to reach the sink. All paths from a task in job j to the sink lead either through a computer or through a node Uj that corresponds to leaving the task unscheduled. Each computer’s outgoing edge has unit capacity, so if there are more tasks than computers some tasks will end up unscheduled.

By controlling the capacities between Uj and the sink we can control fairness by setting the maximum
and minimum number of tasks that a job may leave unscheduled, and hence the maximum and minimum number of computers that the algorithm can allocate to it for
running tasks.

V. EVALUATION

In this section we run a set of applications under different network conditions and compare their performance when scheduled by the queue-based and flow-based algorithms under various policies. We aim to highlight several major points. First, we show that, for our experimental workload, adopting fairness with preemption gives the best overall performance regardless of the scheduler implementation that is used. Second we compare queuebased and flow-based schedulers with each policy, and determine that Quincy, our flow-based implementation, generally provides better performance and lower network utilization. Finally, we show that when the network is a bottleneck, drastically reducing cross-cluster communication has a substantial positive effect on overall throughput and demonstrate that Quincy, unlike the queue-based approach, is easy to tune for different behavior under different
network provisioning setups For all the experiments in this paper we set the waittime factor ! to 0.5 and the cost of transferring 1GB across a rack switch to 1. For most experiments _, the cost of transferring 1GB across the core switch, is set to 2. However for some experiments with policy QFP we
set _ = 20 and these are denoted QFPX.

VI. RELATED WORK

The study of scheduling algorithms for parallel computing resources has a long history [20]. The problem
of assigning jobs to machines can in general be cast as a job-shop scheduling task [16], however theoretical results for job-shop problems tend to be quite general and not directly applicable to specific implementations. Scheduling Parallel Tasks. Ousterhout introduced the concept of coscheduling for multiprocessors [24], where cooperating processes are scheduled simultaneously to improve inter-process communication. In contrast to coscheduling, which runs the same task on all processors at
once using an externally-controlled context switch, implicit scheduling [9] can be used to schedule communicating processes. Neither technique applies in our setting in which running tasks do not communicate. Naik et al. [21] compare three policies for allocating processors to a set of parallel jobs: static, which statically partitions the processors into disjoint sets during system configuration; adaptive, which repartitions the processors when jobs arrive and leave but never changes the allocations during execution; and dynamic, which changes processor allocation as a program is executing. Dynamic partitioning is similar to our approach in that the number of resources allocated to a job can change over time. However Naik et al. assume that the data transfer cost across any two processor is the same and this is not true in our cluster setup. While their overriding goal is to improve system performance, we also care about fairness, especially for small jobs. Stone [28] developed a framework based on network flow for partitioning sequential modules across a set of processors, however his graph construction is quite different to ours and does not have efficient solutions when there are more than two processors.

There is an excellent survey by Norman and Thanisch [22] of the mapping problem for parallel jobs
on multi-processors, however this field of research deals with the offline problem of optimally scheduling a single job and thus does not address fairness. Scheduling in a distributed computing cluster. Bent et
al. [6] describe BADFS which addresses many issues relating to storage management and data locality in a cluster setting similar to ours, but does not attempt to achieve fairness. Amiri et al. [5] describe a system that dynamically migrates data-intensive computations to optimize a cost function, however the details of the optimization are not given, and there is no mention of fairness. Zaharia et al. [33] develop a scheduling algorithm called LATE that attempts to improve the response time of short jobs
by executing duplicates of some tasks. Their approach is complementary to our goal of ensuring fair sharing of resources between jobs. Agrawal et al. [4] demonstrate that Hadoop’s performance can be improved by modifying its scheduling policy to favor shared scans between jobs that read from the same inputs. Their approach relies on correctly anticipating future job arrival rates, which we do not yet incorporate into our scheduling framework, however it may be possible to integrate
their approach with Quincy’s fair-sharing policies. Condor [29] includes a variety of sophisticated policies for matching resources to jobs [25] including cases

where resource requests are not independent [26, 30]. Condor schedulers do not, however, support fine-grain localitydriven resource matching. The closest work to our own in this field is that of Zaharia et al. [32] which addresses a similar problem using a queue-based scheduling approach. Fair-share Scheduling. Proportional share scheduling using priorities [19] is a classical technique but provides only coarse control. Schroeder et al. [27] consider the tradeoff between fairness and performance when scheduling parallel jobs, and Gulati et al. [15] describe a scheduler
called PARDA that implements proportional-share bandwidth allocation for a storage server being accessed by a set of remote clients. However, none of these previous approaches takes account of data locality which is crucial in our setting. As far as we are aware, ours is the first work to model
fair scheduling as a min-cost flow problem.

VII. DISCUSSION

Since our scheduling task maps very naturally to mincost flow, and this mapping leads to extremely effective solutions, it may seem surprising that it has not been previously applied to similar problems in the past. Virtualmachine (VM) placement and migration and many-core operating system scheduling for example both superficially seem closely related. We believe there are several reasons for the lack of
related work. One is that large shared clusters are a relatively recent development. Batch time-sharing systems did not historically have large numbers of independent resources to allocate. Supercomputers and grid clusters have long been massively parallel, but they are typically scheduled using coarse-grained allocation policies and a fixed set of computers is assigned to a job for a lengthy period. Also, there are restrictions on our cost model that make it easy to express as network flow. In particular: We can model the costs of assigning tasks to computers as being independent. For example, we never have a constraint of the form "tasks A and B must execute in the same rack, but I don't care which rack it is." Correlated constraints are very common when a system must schedule processes that communicate with each other while running, and this situation frequently arises in VM placement problems and when executing
other distributed computational models such as MPI. Its difficult to see how to encode correlated constraints directly in a network flow model like Quincy's.The capacities in our graph construction are naturally one-dimensional. In many scheduling settings, each task may consume less than 100% of a computer's resources. Their resource requirements may be modeled as a multi-dimensional quantity, for example the amount of CPU, disk IO and memory the task demands. Multidimensional capacities cannot be easily represented by Quincy's flow network, though as we discuss below we can model fractional capacities. Nevertheless we are optimistic that there will turn out to be numerous scheduling problems that can be encoded using min-cost flow, and that Quincy's effectiveness will spur further research in the area. A limitation of Quincy's current design is that fairness is expressed purely in terms of the number of computers allocated to a job, and no explicit attempt is made to share the network or other resources. As we noted in Section 1, Quincy's ability to reduce unnecessary network traffic does improve overall performance predictability and consequently fairness, however when an application does require cross-cluster communication this requirement does not get modeled by our system. We would like to extend Quincy to take account of network congestion in future work: one interesting idea is to monitor network
traffic and dynamically adjust Quincy's costs to optimize for reduced data transfer only when it is a bottleneck. There are several straightforward extensions to Quincy that we are currently exploring:
_ We can easily support the allocation of multiple tasks to a single computer by increasing the capacity of the edge from each computer to the sink. It may also be advantageous to temporarily assign more tasks to a computer than its capacity allows, and time-slice the tasks, instead of preempting a task that has been running on the computer for a long time. We could get better cluster utilization in some cases
by adaptively setting the job concurrency K, for example opportunistically starting "spare" jobs if cluster
computers become idle, but reducing their worker task allocation to zero if resource contention returns. In our current design, the choice of K and the algorithm determining appropriate allocations Aj for each job j are completely decoupled from the scheduling algorithm. A future design could unify these procedures, making the admission and fairness policies adapt to the current workload and locality constraints.
There is a great deal of scope to incorporate predictions about the running time of jobs and tasks, and the known future workflow of jobs, into our cost model. Pursuing this direction may lead us to move all of the jobs' workflow state machines into Quincy and dispense with root tasks altogether.
In conclusion, we present a new approach to scheduling with fairness on shared distributed computing clusters. We constructed a simple mapping from the fairscheduling problem to min-cost flow, and can thus efficiently compute global matchings that optimize our instantaneous scheduling decisions. We performed an extensive evaluation on a medium-sized cluster, using two network configurations, and confirmed that the global cost-based matching

approach greatly outperforms previous methods on a variety of metrics. When compared to greedy algorithms with and without fairness, it can
reduce traffic through the core switch of a hierarchicalnetwork by a factor of three, while at the same time increasing the throughput of the cluster.

VIII. ACKNOWLEDGMENTS

We would like to thank Martín Abadi, Mihai Budiu, Rama Kotla, Dahlia Malkhi, Chandu Thekkath, Renato Werneck, Ollie Williams and Fang Yu for many helpful comments and discussions. We would also like to thank the SOSP committee for their detailed reviews, and our shepherd Steve Hand for invaluable advice and suggestions. We have no theoretical analysis of the stability of the system, however empirically we have not observed any unexpected reallocations of resources. Also, note that the cost of leaving a task running in its current location monotonically decreases over time, and if ! = 0 or there are no tasks currently unscheduled all other costs in the graph are constant. If a min-cost flow solution is modified
strictly by decreasing the cost of edges that are already at full capacity, the flow assignment remains a mincost solution. Thus when there are no unscheduled tasks the system is guaranteed not to change any scheduling assignment until a task completes or a new task is submitted.

REFERENCES

[1] The hadoop fair scheduler. https://issues. apache.org/jira/browse/HADOOP-3746.
[2] Open MPI. http://www.open-mpi.org/.
[3] Hadoop wiki. http://wiki.apache.org/hadoop/, April 2008.
[4] P. Agrawal, D. Kifer, and C. Olston. Scheduling Shared Scans of Large Data Files. In Proc. VLDB, pages 958–969, 2008.
[5] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic function placement for data-intensive cluster computing.
In Usenix Annual Technical Conference, 2000.
[6] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit Control in a
Batch-Aware Distributed File System. In Proc. NSDI, March 2004.
[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In Proc. OSDI, pages 137–150, December 2004.
[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. Communications of the
ACM, 51(1):107–113, 2008.